**Summary and Reflections Report**

1. **Summary**
   1. Describe your unit testing approach for each of the three features.
      1. To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence. The unit testing approach was intricately aligned with the software requirements, as evidenced by the direct correspondence between the test cases and the expected functionalities outlined in the requirements documentation. Each test case mirrored an anticipated user interaction, providing a thorough validation of the implemented code against the specified requirements.

**Example**:   
@Test

void newTaskTest() {

TaskService service = new TaskService();

service.newTask();

Assertions.assertNotNull(service.getTaskList().get(0).getTaskId());

Assertions.assertNotEquals("INITIAL", service.getTaskList().get(0).getTaskId());

1. Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were **effective** based on the coverage percentage? Code coverage tools were employed to ensure that every line of code within the unit tests was executed. This high coverage percentage indicated that the majority of the code paths were tested, providing confidence in the effectiveness of the JUnit tests to verify the correctness of the implemented functionalities.

**Example**:

@RepeatedTest(4)

void UuidTest() {

TaskService service = new TaskService();

service.newTask();

service.newTask();

service.newTask();

assertEquals(3, service.getTaskList().size());

assertNotEquals(service.getTaskList().get(0).getTaskId(),

service.getTaskList().get(1).getTaskId());

assertNotEquals(service.getTaskList().get(0).getTaskId(),

service.getTaskList().get(2).getTaskId());

assertNotEquals(service.getTaskList().get(1).getTaskId(),

 service.getTaskList().get(2).getTaskId());  
This test method is repeated four times (@RepeatedTest(4)) to ensure that the unique ID generation logic is working consistently across multiple runs. This kind of test helps ensure that the code paths related to generating unique IDs are exercised, contributing to higher code coverage.

1. Describe your experience writing the JUnit tests.
   1. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate. Ensuring technical soundness was a focal point during the writing of JUnit tests. Thorough code reviews were conducted to verify adherence to best practices and JUnit testing framework conventions. An example of technical soundness can be observed in the proper setup and teardown of test environments:

**Example**:   
@SuppressWarnings("deprecation")

@BeforeEach

void setUp() {

id = "1234567890";

description = "The appt object shall have a required description.";

date = new Date(3021, Calendar.JANUARY, 1);

tooLongDescription = "This description is too long for the appointment requirements but good for testing.";

pastDate = new Date(0);

In the setUp method, the technical soundness is demonstrated by properly initializing test data, including setting up a valid date (date), a past date (pastDate), and other attributes. The use of @SuppressWarnings("deprecation") suggests awareness of potential issues with deprecated methods, showing attention to detail in maintaining code quality.

How did you ensure that your code was **efficient**? Cite specific lines of code from your tests to illustrate. Efficiency was emphasized through the use of parameterized tests to cover multiple scenarios with a single test method, reducing redundancy in the test suite. An example of an efficient parameterized test is illustrated below:  
**Example**:  
class ContactServiceTest {

@ParameterizedTest

@CsvSource({

 "John, Doe, 1234567890, New York",

    "Jane, Smith, 9876543210, Los Angeles",

     "Alice, Johnson, 5555555555, Chicago"

})

void newContactWithDetailsTest(String firstName, String lastName,

 String phoneNumber, String address) {

  ContactService service = new ContactService();

  service.newContact(firstName, lastName, phoneNumber, address);

  assertEquals(1, service.getContactList().size());

  assertEquals(firstName, service.getContactList().get(0).getFirstName());

  assertEquals(lastName, service.getContactList().get(0).getLastName());

  assertEquals(phoneNumber, service.getContactList().get(0).getPhoneNumber());

   assertEquals(address, service.getContactList().get(0).getAddress());  
In this example, the newContactWithDetailsTest method is parameterized using the @CsvSource annotation, allowing it to cover multiple scenarios with different sets of parameters. This helps reduce redundancy in the test suite and ensures that various combinations of input parameters are tested efficiently.

1. **Reflection**
   1. Testing Techniques
      1. What were the **software testing techniques** that you employed in this project? Describe their characteristics using specific details. The software testing techniques employed included:

* Unit Testing: Focused on isolating individual components to ensure their correctness.
* Integration Testing: Ensured that different components worked together seamlessly.
* Boundary Value Analysis: Tested values at the extremes of input ranges.  
  **Example**:

1. What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details. Techniques not used in this project include stress testing and performance testing. Stress testing assesses system behavior under extreme conditions, while performance testing evaluates system responsiveness under different workloads.  
   **Example**:
2. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.  
    3A. **Unit testing:** Crucial for validating individual units and ensuring their correctness.  
    3B. **Integration testing:** Verifies the interaction between integrated components, validating the system's overall functionality.  
    3C. **Boundary Value Analysis:** Identifies potential issues at input extremes, helping ensure robustness in handling various inputs.  
   **Example**:
3. Mindset
   1. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution**? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.  
      A cautious mindset was pivotal in identifying potential issues. For instance, in the authentication feature, understanding how user inputs interacted with the database was critical. Appreciating the complexity allowed the creation of meaningful test cases, ensuring thorough validation.  
      **Example**:  
      public void deleteContact(String id) throws Exception { contactList.remove(searchForContact(id)); }  
       Here, the method deleteContact(String id) is responsible for deleting a contact based on its unique ID. The implementation leverages the searchForContact(String id) method to find the contact to be deleted.
   2. Assess the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims. Bias limitation was maintained by approaching the code objectively. If responsible for testing my own code as a developer, bias could arise from overlooking certain paths. To counter this, maintaining a tester's perspective and scrutinizing each feature without preconceived notions was crucial.  
      **Example**:  
      1) The getTaskIdTest, getNameTest, and getDescriptionTest tests check if the getters return the expected values when valid inputs are provided.  
      2) The TaskIdTooLongTest, setTooLongNameTest, and setTooLongDescriptionTest tests validate that the code throws IllegalArgumentException when inputs exceed specified length limits. This is an important aspect of bias limitation, as it prevents unintended behavior when inputs are too long.
   3. Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims. Discipline in commitment to quality is paramount to avoid technical debt. Cutting corners may lead to future issues. Adhering to coding standards, conducting regular code reviews, and prioritizing comprehensive testing are strategies to minimize technical debt. For instance, ensuring thorough testing of all code paths prevents issues from arising later in the development lifecycle.  
      **Example**:  
      private String newUniqueId() {   
      return UUID.randomUUID().toString().substring(0, Math.min(toString().length(), 10)); }  
       In this method, a unique identifier for a new task is generated using UUID.randomUUID(). However, the implementation of creating the unique ID seems flawed and could potentially lead to issues.